Factor Articles - Work in Progress

Chris Double

October 30, 2009

Contents

Foreword

This set of articles is based on postings to my weblog which can be found at `http://www.bluishcoder.co.nz`.

The articles in the first part have been updated to run on the current version of Factor (built from the git repository). The remaining articles still have to be updated. The Factor homepage can be found at `http://www.factorcode.org`.

The LaTeXsource code for these articles can be browsed at:

```
http://www.double.co.nz/cgi-bin/gitweb.cgi?p=factor-articles.git
```

The source code be retrieved with:

```
git clone git://double.co.nz/git/factor-articles.git
```

The template for the LaTeXversion of these articles is based on the original Factor handbook written by Slava Pestov.

Part I

Works in latest Factor

Chapter 1

Cells

Factor has a Cells like mechanism for propagating changes in models to their GUI representation (called Gadgets). It's like a lightweight functional reactive programming system.

1.1 Models

To have a gadget dynamically updated it must reference a 'model' which wraps the value to be displayed. Whenever the data wrapped by the model is changed, a sequence of connected objects are notified of the change.

To create a model, use the `<model>` word, with the value it is wrapping on the top of the stack.

Vocabulary: `models`
Word: `<model> ( value -- model )`

```
USE: models
42 <model> value>> .
  42
```

New models can be created based on the value of existing models. The `<arrow>` word creates a new model, with its value being the result of a quotation applied to the original models value. Its value changes whenever the original model is changed.

Vocabulary: `models`
Word: `<arrow> ( model quot -- arrow )`

```
USE: models
```

```
SYMBOL: a
SYMBOL: b

42 <model> a set
a get [ 2 * ] <arrow> b set
b get activate-model
b get value>> .
  84
10 a get set-model
b get value>> .
  20
```

You'll notice the call to the word `activate-model`. An active model is one that gets updated when the models it depends on change. There is a corresponding `deactivate-model` that can be called to stop changes from propagating.

Vocabulary: `models`
Word: `activate-model ( model -- )`

Vocabulary: `models`
Word: `deactivate-model ( model -- )`

1.2 Gadgets and Models

Gadgets can use models as their underlying data to be displayed. The gadget will automatically be updated when the underlying data changes.

```
"hello"  <model>
dup <label-control> gadget.
"world" over set-model
"!" over set-model
```

Here we create a model containing the text "hello". A label-control gadget is created with this model as the data, and the gadget is printed in the listener with `gadget.`. The label will show "hello" as the text. When the model is updated with new text using `set-model`, the label control displays the new text immediately.

1.3 Updating time example

In this example I create a global value called 'time' that holds a model wrapping the current date and time:

```
USING: calendar models ;


SYMBOL: time
now <model> time set-global
```

The model here wraps the current date and time produced by `now`. To have the value change regularly I start a background thread which sets the value every second:

```
USE: alarms

: update-time ( -- )
  now time get-global set-model ;

[ update-time ] now 1 seconds add-alarm drop
```

The `update-time` words gets the current date and time (using `now`) and sets the model's value to it. An alarm is added to do this every second.

We can confirm that this value is updating by getting the value of the 'time' variable and see that it changes between calls:

```
USE: calendar.format

time get-global value>> timestamp>http-string .
  "Sun, 20 Jan 2008, 02:02:02 GMT"

time get-global value>> timestamp>http-string .
  "Sun, 20 Jan 2008, 02:04:02 GMT"
```

A label gadget that displays this value, and is updated as it changes, can be created and displayed with:

```
USING: ui.gadgets.labels ui.gadgets.panes calendar.format ;

time get-global
[ timestamp>http-string ] <arrow> <label-control>
gadget.
```

Notice I'm using `<arrow>` to take the time value and convert it into a string. A `<label-control>` can only display strings, so it has the filtered value as its data. The gadget displayed on the listener will update automatically as the time changes every second.

Chapter 2

Search and Replace with PEGs

The Parsing Expression Grammar library has code to allow search and replace using parsers. The idea was to be able to do similar things with parser combinators that regular expressions are commonly used for. The words are in the `peg.search` vocabulary.

Vocabulary: `peg.search`
Word: `search ( string parser -- seq )`

The 'search' word takes a string and a parser off the stack. It returns a sequence of all substrings in the original string that were successfully parsed by the parser. The result returned in the sequence is the AST produced by the parser. For example:

```
USING: peg peg.ebnf peg.parsers peg.search ;

: bold ( -- parser )
  <EBNF rule="*" (!("*") .)+:s "*" => [[ s >string ]] EBNF> ;

"hello *world* from *factor*" bold search .
 => V{ "world" "factor" }

"one 100 two 300" 'integer' search .
 => { 100 300 }

"one 100 two 300" 'integer' [ 2 * ] action search .
 => { 200 600 }
```

Vocabulary: `peg.search`
Word: `replace ( string parser -- result )`

The 'replace' word takes a string and parser off the stack. It returns the original string with all substrings that successfully parse by the parser replaced by the result of that parser.

```
"Hello *World*" bold [
  "<strong>" "</strong>" surround
] action replace .
 => "Hello <strong>World</strong>"
```

Chapter 3

Channels

Rob Pike gave a talk at Google about the NewSqueak programming language, and specifically how it's concurrency features work. NewSqueak uses channels and is based on the concepts of Communicating Sequential Processes.

Factor has a 'channels' vocabulary that lets you do similar things.

Vocabulary: `channels`
Word: `<channel> ( -- channel )`
Word: `to ( value channel -- )`
Word: `from ( channel -- value )`

The `<channel>` word creates a channel that threads can send data to and receive data from. The `to` word sends data to a channel. It is a synchronous send and blocks waiting for a receiver if there is none. The `from` word receives data from a channel. It will block if there is no sender.

There can be multiple senders waiting, and if a thread then receives on the channel, a random sender will be released to send its data. There can also be multiple receivers blocking. A random one is selected to receive the data when a sender becomes available.

Here is the 'counter' example ported from NewSqueak:

```
USING: channels fry threads ;

: (counter) ( channel n -- )
  [ swap to ] [ 1 + (counter) ] 2bi ;

: counter ( channel -- )
  2 (counter) ;

: counter-test ( -- n1 n2 n3 )
```

```
  <channel> dup '[ _ counter ] "counter" spawn drop
  [ from ] [ from ] [ from ] tri ;

counter-test . . .
 => 2
    3
    4
```

Given a channel, the `counter` word will send numbers to it, incrementing them by one, starting from two. `counter-test` creates a channel, spawns a process to run `counter`, and then receives a few values from the channel.

Chapter 4

Distributed Channels

Remote Channels are distributed channels that allow you to access channels in separate Factor instances, even on different machines on the network.

Vocabulary: `channels.remote`
Word: `<remote-channel> ( node id -- remote-channel )`
Word: `publish ( channel -- id )`

A channel can be made accessible by remote Factor nodes using the `publish` word. Given a channel this will return a unique identifier that can be used by remote nodes to use the channel. For example:

```
USING: fry threads channels channels.remote ;

: (counter) ( channel n -- )
  [ swap to ] [ 1 + (counter) ] 2bi ;

: counter ( channel -- )
  2 (counter) ;

<channel> dup '[ _ counter ] "counter" spawn drop publish .
 => 12345678901234567890.....
```

Remote channels are implemented using distributed concurrency so you must start a node on the Factor instance you are using. This is done with 'start-node' giving the hostname and port:

Vocabulary: `concurrency.distributed`
Word: `start-node ( port -- )`

```
USE: concurrency.distributed
9000 start-node
```

Once this is done all published channels become available. Note that the hostname and port must be accessible by the remote machine so it can connect to send the data you request.

From a remote node you can create a `<remote-channel>` which contains the hostname and port of the node containing the channel, and the identifer of that channel.

You can use 'from' and 'to' on the remote channel exactly as you can on normal channels. The data is marshalled over the network using the serialization library.

```
USING: io.sockets ;

"127.0.0.1" 9000 <inet4> 1234... <remote-channel> from .
```

One way of setting up remote channel services is to serialize an instance of a `<remote-channel>` for a published channel and make it available on an HTTP server. The remote nodes can retrieve this via HTTP, deserialize it and use it. You can test this on a local machine by running two factor instances. In instance 1:

```
USING:
  fry
  io.encodings.binary
  threads
  concurrency.distributed
  channels
  channels.remote
  serialize
;

9000 start-node

: (counter) ( channel n -- )
  [ swap to ] [ 1 + (counter) ] 2bi ;

: counter ( channel -- )
  2 (counter) ;

<channel> dup '[ _ counter ] "counter" spawn drop
publish local-node get swap <remote-channel>
"counter.ser" binary [ serialize ] with-file-writer
```

This creates a channel that returns incrementing integer numbers, as per our previous examples. A `<remote-channel>` is created with the published identifier for this channel and our node address. This is serialized to a file called "counter.ser". A remote node can deserialize this file and use it to access the channel immediately. In factor instance 2:

```
USING:
  io.encodings.binary
  threads
  concurrency.distributed
  channels
  channels.remote
  serialize
;

9001 start-node
"counter.ser" binary [ deserialize ] with-file-reader
dup from .
 => 2
dup from .
 => 3
```

Chapter 5

Pattern Matching

Given a Factor sequence or primitive type you can pattern match and create bindings to symbols with a special format. Any symbol that begins with a '?' character is used as a pattern match variable, and binds to the value in a matching sequence. Here's a simple example:

Vocabulary: `match`
Word: `match ( value1 value2 -- bindings )`

```
USE: match

MATCH-VARS: ?a ?b ?c ;

{ 1 2 3 } { ?a ?b ?c } match .
  H{ { ?a 1 } { ?b 2 } { ?c 3 } }
```

The two sequences match in that they are both sequences with three items. So the result is a hashtable containing the pattern match variables as keys and the values of those variables in the second sequence as the value. If items in the sequence are not pattern match variables then they must match exactly:

```
{ 1 2 3 } { 1 2 ?a } match .
  H{ { ?a 3 } }

{ 1 2 3 } { 2 2 ?a } match .
  f
```

The second example doesn't match as the first element in both sequences is not the value '2'.

Matching works recursively too:

```
{ 1 2 { 3 4 } { 5 6 } } { 1 2 ?a { ?b ?c } } match .
  H{ { ?a { 3 4 } } { ?b 5 } { ?c 6 } }
```

This type of pattern matching is very useful for deconstructing messages sent to distributed factor objects. But even more useful is to be able to have something like `cond` but working directly with patterns. This is what `match-cond` does. It takes a sequence and an array of arrays. Each sub-array contains the pattern to match the sequence against, and a quotation to execute if that pattern matched. The quotation is executed with the hashtable result of the pattern match bound in the current namespace scope, allowing easy retrieval of the pattern match variables.

Vocabulary: `match`
Word: `match-cond ( assoc -- )`

It sounds complex but is actually very easy in practice. Here what an example 'counter' process might look like using `match-cond`:

```
USING: match concurrency.messaging ;

SYMBOL: increment
SYMBOL: decrement
SYMBOL: get
MATCH-VARS: ?value ?from ?tag ;

: counter ( value -- )
  receive {
    { { increment ?value } [ ?value + ] }
    { { decrement ?value } [ ?value - ] }
    { { get ?from }        [ dup ?from send ] }
    { _                    [ "Unmatched message" print flush ] }
  } match-cond counter ;
```

This is a process that keeps track of a count value. You can send messages to increment or decrement the count by an amount, or to get the value and send it back to the calling process. So sends/receives look like:

```
USING: threads ;

[ 0 counter ] "counter" spawn
{ increment 5 } over send
{ decrement 10 } over send
[ get , self , ] { } make swap send receive .
  -5
```

One thing to be aware of with `match-cond` is that calls in the quotation side of a condition are not tail recursive. So the following looks equivalent to the previous

counter example but is not tail recursive and will consume call stack space on each message, eventually failing when it overflows:

```
: counter ( value -- )
  receive {
    { { increment ?value } [ ?value + counter ] }
    { { decrement ?value } [ ?value - counter ] }
    { { get ?from }        [ dup ?from send counter ] }
    { _                    [ "Unmatched message" print flush counter ] }
  } match-cond ;
```

Pattern matching on tuples, and the repeating presence of match variables also works:

```
TUPLE: person first last ;
MATCH-VARS: ?first ?last ;

T{ person f "chris" "double" } T{ person f ?first ?last } match .
  H{ { ?first "chris" } { ?last "double" } }

{ "one" "two" "one" "two" } { ?a ?b ?a ?b } match .
  H{ { ?a "one" } { ?b "two" } }

{ "1" "two" "one" "two" } { ?a ?b ?a ?b } match .
  f
```

Part II

Need to be updated

Chapter 6

Embedded Domain Specific Languages in Factor

I've been doing some experimenting with the emedded grammar code I wrote for Factor, trying to make it easier to use and a bit more useful for real world projects.

My inspiration for the changes has been seeing the kinds of things OMeta can do and the examples in the Steps towards the reinvention of programming from the Viewpoints Research Institute.

The original parsing expression grammar code (in the vocab 'peg') produced a data structure composed of Factor tuples that was interpreted at runtime via a call to the word 'parse'. It still has the data structure of tuples but now it can be compiled into Factor quotations, which are then compiled to native machine code via the Factor compiler. The 'parse' word calls 'compile' on the datastructure and calls it.

I created a parsing word that allows you to embed the peg expression directly in code, have it compiled to a quotation at parse time, and then called at runtime. Usage looks like:

```
"1+2" [EBNF expr=[0-9] '+' [0-9] EBNF] call
```

The older peg code had an `<EBNF ... EBNF>` embedded language and each rule in the language was translated to a Factor word. That's now changed to an EBNF: definition. An example:

```
EBNF: expr
digit    = [0-9]            => [[ digit> ]]
number   = (digit)+         => [[ 10 digits>integer ]]
value    =   number
           | ("(" exp ")")  => [[ second ]]
```

```
fac      =   fac:x "*" value:y  => [[ drop x y * ]]
           | fac:x "/" value:y  => [[ drop x y / ]]
           | number

exp      =   exp:x "+" fac:y    => [[ drop x y + ]]
           | exp:x "-" fac:y    => [[ drop x y - ]]
           | fac
;EBNF
```

This creates a word, 'expr', that runs the last rule in the embedded language (the 'exp' rule in this case) on the string passed to it:

```
"1+2*3+4" expr parse-result-ast .
 => 11
```

If you've used peg.ebnf before you'll see some new features in this code:

- You can do character ranges using code like `[a-zA-Z]` to match upper and lowercase characters, etc.
- Factor quotations can be embedded to process the results of choices. Anything between the `[[ ... ]]` will be run when that choice succeeds and the result put in the abstract syntax tree for that rule. The default AST produced by the rule is on the stack of the quotation. The example above drops this in some cases, and transforms it in others.
- Rule elements can have variable names suffixed to it and these can be referenced in the action quotations. This is implemented using locals. This can be seen in EBNF code like this:

  ```
  exp:x "+" exp:y => [[ drop x y + ]]
  ```

  Usually that rule produces an AST consisting of a 3 element sequence, each element being the AST produced by the rules elements. The action quotation is transformed into:

  ```
  [let* | x [ 0 over nth ]
          y [ 2 over nth ] |
    drop x y +
  ] with-locals
  ```

  This is efficient and makes the grammar easier to read.

- Another major new feature is grammars now handle direct and indirect left recursion. I implemented this from the VPRI paper Packrat Parsers Can Support Left Recursion. It makes converting existing grammars to peg grammars much easier.

- Semantic actions have been added. These are like normal `[[ ... ]]` actions except they have stack effect `( ast -- bool )`. Given an abstract syntax tree from the preceding element, it should return a boolean indicating whether the parse succeeded or not. For example:

  ```
  odd= [0-9] ?[ digit> odd? ]?
  ```

- Some of the syntax has changed. Previously `{ ... }` was used for repeating zero or more times and `[ ... ]` was for optional. Now I use `(...)*`, `(...)+`, `(...)?`, for zero or more, one or more, and optional respectively. The dot `(.)` is used to match anything at that point. Terminals can be enclosed in single or double quotations.

- There is a 'rule' word that can be used to get a single rule from a compiled grammar:

  ```
  EBNF: foo
  number=([0-9])+
  expr  = number '+' number
  ;EXPR

  "1+2" foo parse-result-ast => { "1" "+" "2" }
  "1+2" "number" \ foo rule parse parse-result-ast => "1"
  ```

  Notice the 'rule' word returns the parser object rather than the compiled quotation. This is useful for testing and further combining with other parsers.

These changes are in the main Factor repository. There is the peg.pl0 and peg.expr vocabs as examples. The peg.ebnf code is in an experimental state and is likely to change a lot until I'm satisfied with it so be aware that it might not be wise to use it in stable code unless you're happy with tracking the changes. I welcome feedback and ideas though.

One feature I'm currently working on but haven't put in the main repository yet is the ability to use the embedded grammar DSL to operate over Factor arrays and tuples. This allows writing an embedded grammar to produced an AST, and another embedded grammar to transform that AST into something else. Here's what code to transform an AST currently looks like:

```
TUPLE: plus lhs rhs ;

EBNF: adder
```

```
num   = . ?[ number? ]?
plus  = . ?[ plus? ]? expr:a expr:b => [[ drop a b + ]]
expr  =  plus | num
;EBNF

T{ plus f 1 T{ plus f 2 3 } } adder parse-result-ast .

=> 6
```

This uses features I've already discussed, like semantic actions, to detect the type of the object. The difference is that the parser produced by EBNF: operates not on strings, but on an abstract sequence type that is implemented for strings, sequence, and tuples. I'm still playing around with ideas for this but I think it'll be a useful way to reuse grammars and string them together to perform tasks.

Chapter 7

Parsing JavaScript

I've made some more changes to the Parsing Expression Grammar library in Factor. Most of the changes were inspired by things that OMeta can do. The grammar I used for testing is an OMeta-JS grammar for a subset of JavaScript. First the list of changes:

- Actions in the EBNF syntax receive an AST (Abstract Syntax Tree) on the stack. The action quotation is expected to have stack effect `( ast -- ast )`. It modifies the AST and leaves the new version on the stack. This led to code that looks like this:

  ```
  expr = lhs '+' rhs => [[ first3 nip + ]]
  ```

  Nothing wrong with that, but a later change added variables to the EBNF grammar to make it more readable:

  ```
  expr = lhs:x '+' rhs:y => [[ drop x y + ]]
  ```

  Code that uses variables a lot end up with a lot of 'drop' usage as the first word. I made a change recommended by Slava to have the action add the drop automatically depending on the stack effect of the action. So now this code works:

  ```
  expr = lhs:x '+' rhs:y => [[ x y + ]]
  ```

  So now if you use variables in a rule, the stack effect of the action should be `( -- ast )`. If you don't, it should be `( ast -- ast )`.

- I added a way for one EBNF parser to call rules defined in another. This allows creating grammars which are hybrids of existing parsers. Or just to reuse common things like string handling expressions. These calls are called 'foreign' calls and appear on the right hand side of a rule in angle brackets. Here is a parser that parses strings between double quotation marks:

```
EBNF: parse-string
StringBody = (!('"') .)*
String= '"' StringBody:b '"' => [[ b >string ]]
;EBNF
```

To call the 'String' rule from another parser:

```
EBNF: parse-two-strings
TwoStrings = <foreign parse-string String> <foreign parse-string S
;EBNF
```

The `<foreign>` call in this example takes two arguments. The first is the name of an existing EBNF: defined parser. The second is the rule in that parser to invoke. It can also be used like this:

```
EBNF: parse-two-strings
TwoString = <foreign parse-string> <foreign parse-string>
;EBNF
```

If the first argument is the name of an EBNF: defined parser and no second argument is given, then the main rule of that parser is used. The main rule is the last rule in the parser body. A final way foreign can be used:

```
: a-token ( -- parser ) "a" token ;

EBNF: parse-abc
abc = <foreign a-token> 'b' 'c'
;EBNF
```

If the first argument given to foreign is not an EBNF: defined parser, it is assumed that it has stack effect `( -- parser )` and it will be called to return the parser to be used.

- It is now possible to override the tokenizer in an EBNF defined parser. Usually the sequence to be parsed is an array of characters or a string. Terminals in a rule match successive characters in the array or string. For example:

```
EBNF: foo
rule = "++" "--"
;EBNF
```

This parser when run with the string "++–" or the array { `CHAR: + CHAR: + CHAR: - CHAR: -` } will succeed with an AST of { `"++" --"` }. If you want to add whitespace handling to the grammar you need to put it between the terminals:

```
EBNF: foo
space = (" " | "\r" | "\t" | "\n")
spaces = space* => [[ drop ignore ]]
rule = spaces "++" spaces "--" spaces
;EBNF
```

In a large grammar this gets tedious and makes the grammar hard to read. Instead you can write a rule to split the input sequence into tokens, and have the grammar operate on these tokens. This is how the previous example might look:

```
EBNF: foo
space = (" " | "\r" | "\t" | "\n")
spaces = space* => [[ drop ignore ]]
tokenizer = spaces ( "++" | "--" )
rule = "++" "--"
;EBNF
```

'tokenizer' is the name of a built in rule. Once defined it is called to retrieve the next complete token from the input sequence. So the first part of 'rule' is to try and match "++". It calls the tokenizer to get the next complete token. This ignores spaces until it finds a "++" or "–". It is as if the input sequence for the parser was actually { "++" "--'' } instead of the string "++–". With the new tokenizer "...." sequences in the grammar are matched for equality against the token, rather than a string comparison against successive items in the sequence. This can be used to match an AST from a tokenizer:

```
TUPLE: ast-number value ;
TUPLE: ast-string value ;

EBNF: foo-tokenizer
space = (" " | "\r" | "\t" | "\n")
spaces = space* => [[ drop ignore ]]

number = [0-9]* => [[ >string string>number ast-number boa ]]
string =  => [[ ast-string boa ]]
operator = ("+" | "-")

token = spaces ( number | string | operator )
tokens = tokenizer*
;EBNF

ENBF: foo
tokenizer = <foreign foo-tokenizer token>

number = . ?[ ast-number? ]? => [[ value>> ]]
string = . ?[ ast-string? ]? => [[ value>> ]]
```

```
rule = string:a number:b "+" number:c => [[ a b c + 2array ]]
;EBNF
```

In this example I split the tokenizer into a separate parser and use 'foreign' to call it from the main one. This allows testing of the tokenizer separately:

```
"123 456 +" foo-tokenizer ast>> .
=> { T{ ast-number f 123 } T{ ast-number f 456 } "+" }
```

The '.' EBNF production means match a single object in the source sequence. Usually this is a character. With the replacement tokenizer it is either a number object, a string object or a string containing the operator. Using a tokenizer in language grammars makes it easier to deal with whitespace. Defining tokenizers in this way has the advantage of the tokenizer and parser working in one pass. There is no tokenization occurring over the whole string followed by the parse of that result. It tokenizes as it needs too. You can even switch tokenizers multiple times during a grammar. Rules use the tokenizer that was defined lexically before the rule. This is usefull in the JavaScript grammar:

```
EBNF: javascript
tokenizer         = default
nl                = "\r" "\n" | "\n"
tokenizer         = <foreign tokenize-javascript Tok>
...

End                = !(.)
Name               = . ?[ ast-name?   ]?   => [[ value>> ]]
Number             = . ?[ ast-number? ]?   => [[ value>> ]]
String             = . ?[ ast-string? ]?   => [[ value>> ]]
RegExp             = . ?[ ast-regexp? ]?   => [[ value>> ]]
SpacesNoNl         = (!(nl) Space)* => [[ ignore ]]
Sc                 = SpacesNoNl (nl | &("}") | End)| ";"
...
```

Here the rule 'nl' is defined using the default tokenizer of sequential characters ('default' has the special meaning of the built in tokenizer). This is followed by using the JavaScript tokenizer for the remaining rules. This tokenizer strips out whitespace and newlines. Some rules in the grammar require checking for a newline. In particular the automatic semicolon insertion rule (managed by the 'Sc' rule here). If there is a newline, the semicolon can be optional in places.

```
"do" Stmt:s "while" "(" Expr:c ")" Sc    => [[ s c ast-do-while boa ]]
```

Even though the JavaScript tokenizer has removed the newlines, the 'nl' rule can be used to detect them since it is using the default tokenizer. This allows grammars to mix and match the tokenizer as required to make them more readable.

The JavaScript grammar is in the peg.javascript.parser vocabulary. The tokenizer is in peg.javascript.tokenizer. You can run it using the 'parse-javascript' word in peg.javascript:

```
USE: peg.javascript
"var a='hello'; alert(a);" parse-javascript ast>> pprint
T{ ast-begin f
  V{
      T{ ast-var f "a" T{ ast-string f "hello" } }
      T{ ast-call f
        T{ ast-get f "alert" } V{ T{ ast-get f "a" } } }
  }
}
```

Chapter 8

Partial Continuations

I've written some partial continuation support and put it in the Factor contrib directory. It implements bshift and breset as outlined by Oleg's post on comp.lang.scheme.

It should be relatively easy to convert Oleg's Scheme examples to Factor. Just remember that the partial continuation has stack effect (a – b) and the quotations passed to bshift and breset have stack effect (pcc – v) and (– v) respectively.

'breset' marks the scope of the partial continuation. If 'bshift' is not used then the value returned by the quotation is left on the stack:

```
[ drop 5 ] breset
 => 5
```

An example with 'bshift':

```
[
 1 swap [ 5 swap call ] bshift +
] breset
 => 6
```

In this case the partial continuation passed to the 'bshift' quotation represents the computation '1 X +' where 'X' is replaced by the value passed to the partial continuation. In this case 5, resulting in a result of 6. Additional calls can be made to the same continuation:

```
[
 1 swap [ 5 over call swap call ] bshift +
] breset
 => 7
```

This calls the '1 X +' partial continuation twice. First with '5' returning the value '6'. Which is then passed to it again, returning '7'. The partial continuation does not need to be called. Values that 'fall through' cause the result to be returned from the 'breset' quotation:

```
[
 1 swap [ drop 5 ] bshift +
] breset
 => 5
```

Here's a fun example translated from Oleg's posting. The following 'range' function has some interesting properties:

```
 : range ( r from to -- )
 rot [ ( from to pcc -- )
   -rot [ over + pick call drop ] each 2drop f
 ]  bshift ;
```

It uses the standard 'each' call on a 'from' and 'too' number to call a quotation on each number between 'from' and 'too' inclusive. The quotation called is the partial continuation provided by 'bshift'. This can be used in code like:

```
[ 1 5 range . ] breset drop
 => 1
    2
    3
    4
    5
```

For each item in the range it executed the partial continuation which is 'X .', printing that item in the range. So given a function that knows nothing about the special capability of 'range' can still work with it. The following prints the first five factorials:

```
: fact ( n -- n ) dup 1 = [ 1 ] [ dup 1 - fact ] if * ;

[ 1 5 range fact . ] breset drop
 => 1
   2
   6
   24
   120
```

I'm not sure how useful delimited continuations are in Factor but it gives something to play with to see how they work.

I've made some changes to the parser combinator library I wrote for Factor. The changes were to make the usage of the library to be more consistent with good Factor style.

The first change was to ensure that items on the stack are accessable in an intuitive manner from within the quotations passed to breset and bshift. When writing combinators in Factor it's important that the internals of the combinator implementation do not affect the callers view of items on the stack. For example:

```
20 [ drop 2 * ] breset
```

Recall that the quotation passed to breset has stack effect (r – v). Once the 'r' is dropped the '*' should be expected to operate on the '2' and the '20'. The '20' is accessable as if it was directly under the 'r' on the stack from within the quotation.

In a prior implementation of breset I called the quotation using 'call' after creating a 'dup' of the 'r':

```
: breset
  ... ( quot r -- )
  dup rot call ( r v -- )
```

The problem with this is that from within the quotation there is an extra 'r' above items on the stack before the quotation is called:

```
20 [ drop 2 * ] breset
```

The stack on entry to the quotation here is (20 r r –). Changing the breset implementation to use 'keep' instead of call fixes this problem:

```
: breset
  ... ( quot r -- )
  swap keep ( v r -- )
```

Notice also that the return item from the quotation, 'v', is now below the 'r' and I didn't need to 'dup' it. In general, usage of words like 'keep' will enable combinators to more intuitively access stack items from outside the quotation.

Another way of solving this problem is to use the return stack words '¿r' and 'r¿' to move items temporarily off the stack so that the quotation being called is imediately above the relevant stack items supplied by the caller.

The second major change was to remove the use of 'with-scope' and namespaces to store the continuation delimiter. I previously stored this in a 'mark' and 'mark-old' variable. Now the implementations of breset and bshift manage these on the stack.

Storing variables in namespaces is seductive. It enables you to avoid sometimes complicated stack management by giving you named variables. Unfortunately it comes with a price. When setting the value of a variable in a namespace it is stored in the top most namespace on the namespace stack. This can be seen with code like this:

```
SYMBOL: myvar
5 myvar set
myvar get .
 => 5
10 [ myvar set ] keep drop myvar get .
 => 10
```

As expected setting the myvar variable in the quotation passed to ‘keep’ results in the global myvar value being set to 10. But if this is run inside a ‘with-scope’ you get caught by the fact that a new namespace is at the top of the stack:

```
myvar get .
  => 10
20 [ [ myvar set ] with-scope ] keep drop myvar get .
  => 10
```

Notice the value is still 10. This is because the variable is set in the new namespace which is dropped off the namespace stack when the ‘with-scope’ exits. Normally you would be aware of this when you write code, but if you use a combinator that uses ‘with-scope’ in its implementation, and it then calls your quotation from within that scope then all your variable sets will be lost at the end of the combinator call.

This may be desired, and the reason why the combinator is in a scope, but for many cases it’s not the desired behaviour. So as a general rule I try to avoid using variables and with-scope in combinator implementations.

There was also a minor bug in my ‘range’ example in that it didn’t give the correct range if the starting number was anything but ‘1’. The corrected ‘range’ implementation is:

```
: range ( r from to -- n )
 over - 1 + rot [
   -rot [ over + pick call drop ] each 2drop f
 ]  bshift 2nip ;
```

Note the ‘2nip’ at the end. ‘bshift’ and ‘breset’ operate like other continuation combinators in that they restore the stack to what they were before they were called. In this case the ‘from’ and ‘to’ were on the stack and we need to ‘nip’ them off. Simple usage works as before:

```
[ 1 5 range . f ] breset drop
 => 1
    2
    3
    4
    5
```

Nested usage works correctly now:

```
[ dup 1 3 range swap 10 12 range + . f ] breset drop
 => 11
    12
    13
    12
    13
    14
    13
    14
    15
```

It can be hard to reason about what usage of words like ‘range’ does. Think of it as returning a single value ‘n’, and calling the breset multiple times with the value of ‘n’ for each ‘n’ in the range. So you’ll see that after the first ‘range’ call I ‘swap’ to swap the result of the range and get the continuation mark passed to breset back to the top of the stack to call the second ‘range’.

You’ll notice that the result of calling these snippets always returns ‘f’ on the stack. This is because the range implementation leaves ‘f’ on the stack at the end of the bshift quotation. This results in ‘breset’ exiting with that value.

A question was raised on the Factor mailing list about CLU-style iterators. An example of the usage from one of the links in that message is:

```
sum: INT := 0;
loop sum := sum + 1.upto!(10); end;
#OUT + sum + '\n'; -- Prints sum of integers from 1 to 10
```

This has a very direct translation in Factor using range and breset as:

```
SYMBOL: sum
0 sum set
[ 1 10 range sum get + sum set f ] breset drop
sum get .
 => 55
```

Chapter 9

Distributed Concurrency

I've just added simple distributed message passing support to the concurrency library. Processes now belong to 'nodes'. These are individual Factor instances running on a machine. Messages can be sent between Factor nodes using the same foramt as sending from processes within a Factor instance.

So far the support I've added is very basic and not at all optimized but it works. Messages can be any Factor type supported by the serialization library.

I've extended the serialization library by added a serializer for local processes that serializes it as a 'remote-process'. This holds the node details (hostname and port) and the process identifier (known as the pid). This allows you to send a local process to a remote process, and that remote process can send a reply back to the local one by sending a message to the remote-process object it receives.

A possible future extension to this might be to serialize proxies for other types. For example, sending a stream in a message can serialize it as a proxy stream so that writes to it from the remote system are sent back to the stream on the local system.

The current state of the system is in my repository:

```
darcs get http://www.bluishcoder.co.nz/repos/factor
```

The 'start-node' word is used to start up a TCP listener to handle requests from remote processes. This is required for distributed message passing to work. If you don't use 'start-node' only local message sends will be enabled. Here's a simple example that sends a message to a remote process, and it sends a reply back to the caller:

```
#! On Machine 1
"concurrency" require
USE: concurrency
```

```
"machine1.com" 9000 start-node

: process1 ( -- )
  receive "Message Received!" reply process1 ;

[ process1 ] spawn "process1" swap register-process
```

This starts the node up with hostname ‘machine1.com’ on port 9000. A word called ‘process1’ is defined that blocks until it receives a message (either from another local process or a remote process) and replies to the message sender with the string ”Message Received!”. It then calls itself to restart the blocking on message receive.

This word is spawned as a process and registered in that nodes global register of named processes as ‘process1’.

```
#! On Machine 2
"concurrency" require
USE: concurrency

"machine2.com" 9000 start-node

[ "machine1.com" 9000  "process1"
  "test-message" swap send-synchronous .
] spawn
```

This code, run in a Factor instance on Machine 2, starts a node with hostname ‘machine2.com’ on port 9000. It spawns a process which sends a message to the process named ‘process1’ on the node at hostname machine1.com, port 9000.

This example sends a synchronous message. It is the equivalent of Termite Scheme’s ‘!?’ operator or Joe Armstrongs ‘!!’ proposed Erlang extension - basically an RPC call. The message is sent to process one and blocks waiting for a reply to that specific message. On the reply it displays it (using ‘.’) which results in ‘Message Received!’ being printed.

Asynchronous sends work too. For example, a logger process on machine 1:

```
#! On Machine 1

: logger ( -- )
  receive print logger ;

[ logger ] spawn "logger" swap register-process
```

Messages can be sent to this process from any machine with:

```
#! On Machine 2

[ "machine1.com" 9000  "logger"
  "Log Message!" swap send
] spawn
```

After this message send ‘Log Message!’ will be printed on machine 1.

Messages can be sent to named processes, registered in the nodes global registry, as these examples show, or they can be sent to any process given the pid - a unique identifier for that process. They can also be sent to remote processes given a deserialized reference to the process object. You could store on a file system or web server deserialized references to important processes that clients can send messages to.

I’m still working on the public API and making the performance better. Currently all message sends open and close the TCP socket to the remote server per message. There is also no security. The server connection for the node is accessable by anyone. Initially I may follow the Erlang ‘magic cookie’ approach to prevent unauthorised message sends but keen to look at better options.

My motivation for working on this is to add to my in-progress web framework the ability to have the server side processes distributed across Factor instances or machines. This is one way to enable Factor to use multiple processors in a machine for example.

Chapter 10

Lazy

I originally wrote the Lazy Lists library in Factor to support the Parser Combinators library. It was one of my first libraries in Factor and was probably not very well written. Matthew Willis worked on it for the recent Factor releases to get it working with the removal of the standard list datatype and made it a lot better in places.

Both the lazy lists and parser combinators worked by building up quotations manually to defer computation, and having these quotations called when required. This unfortunately made the library a bit difficult to debug and understand. While using the parser combinators library for a recent project I found it quite hard to work some things out and found a couple of bugs in the parser combinators - and areas where the lazy lists weren't lazy enough.

To resolve the issues I rewrote the lazy lists library in a slightly different style and plan to redo the parser combinators in a similar manner. Instead of building up quotations I now use tuples to hold the information, with generic functions to call to process them.

I created a generic function based protocol for lists. It has three generic functions:

```
GENERIC: car  ( cons -- car )
GENERIC: cdr  ( cons -- cdr )
GENERIC: nil? ( cons -- bool )
```

A 'cons' tuple for normal non-lazy lists, with the obvious implementation was the starting point:

```
TUPLE: cons car cdr ;
M: cons car ( cons -- car )
    cons-car ;

M: cons cdr ( cons -- cdr )
```

```
    cons-cdr ;

: nil ( -- cons )
  T{ cons f f f } ;

M: cons nil? ( cons -- bool )
    nil eq? ;
```

This gives the functionality of ordinary lists that used to exist in Factor. To make actual lazy lists I use the ‘force’ and ¡promise¿ words from Matthew’s lazy list rewrite. A ¡promise¿ is a wrapper around a quotation that calls that quotation when ‘force’ is called on it, and memoizes the value to return directly on later ‘force’ calls. A lazy list has a promise for both the ‘car’ and ‘cdr’:

```
: lazy-cons ( car cdr -- promise )
    >r <promise> r> <promise> <cons>
    T{ promise f f t f } clone [ set-promise-value ] keep ;

M: promise car ( promise -- car )
  force car force ;

M: promise cdr ( promise -- cdr )
  force cdr force ;

M: promise nil? ( cons -- bool )
  force nil? ;
```

Notice that the lazy list itself is a promise. And the methods are specialized on the ¡promise¿ type. So not only are the ‘car’ and ‘cdr’ promises, but so is the list itself. The ‘cdr’ of a lazy list is a promise, which when forced, returns something that conforms to the list protocol. This means parts of the list can be lazy, and parts non-lazy

A lazy map operation is the first word I implemented on top of this generic protocol. Previous implementations used quotations that were automatically generated and were quite complicated. In this new implementation I created a ¡lazy-map¿ tuple that implemented the list protocol which turned out to be much simpler:

```
TUPLE: lazy-map cons quot ;

: lmap ( list quot -- result )
    over nil? [ 2drop nil ] [ <lazy-map> ] if ;

M: lazy-map car ( lazy-map -- car )
  [ lazy-map-cons car ] keep
  lazy-map-quot call ;
```

```
M: lazy-map cdr ( lazy-map -- cdr )
  [ lazy-map-cons cdr ] keep
  lazy-map-quot lmap ;

M: lazy-map nil? ( lazy-map -- bool )
  lazy-map-cons nil? ;
```

Basically the ‘lmap’ call itself just returns a ¡lazy-map¿ which contains the quotation and list to map over. Calls to ‘car’ will call the quotation on the head of the list, while ‘car’ returns a new ¡lazy-map¿ that holds the same quotation and the ‘cdr’ of the original list.

The ‘take’ operation returns the first ‘n’ items in the list (whether lazy or not). It’s lazy implementation is:

```
TUPLE: lazy-take n cons ;

: ltake ( n list -- result )
    over zero? [ 2drop nil ] [ <lazy-take> ] if ;

M: lazy-take car ( lazy-take -- car )
  lazy-take-cons car ;

M: lazy-take cdr ( lazy-take -- cdr )
  [ lazy-take-n 1- ] keep
  lazy-take-cons cdr ltake ;

M: lazy-take nil? ( lazy-take -- bool )
  lazy-take-n zero? ;
```

Given a word ‘naturals’ to return an infinite list of natural numbers, the squares of the first 10 numbers can be returned as:

```
naturals [ dup * ] lmap 10 swap ltake
```

The result is a lazy list itself. No actual computation is done yet until ‘car’ or ‘cdr’ is called on the result.

More code implements ‘subset’ to filter out items from the lists, append lazy lists in a lazy manner, etc. The squares of all odd natural numbers are expressed as:

```
naturals [ odd? ] lsubset [ dup * ] lmap
```

It would be possible to add to this to implement the protocol for other things like lines in a file. This way you can lazily read through a large file line by line. Although this could be done in the previous lists code I think this approach makes things easier to read. As part of the refactoring I also wrote integrated Factor help for all the non-internal words.

I'm not sure how 'concatenative' this approach is or whether it's the right approach for 'stack' based languages but it seems to work well for Factor. I'll work on the parser combinators, fitting them into a similar style, and see how it works out with the current project.

Chapter 11

Parsers

I was asked on IRC how to use parser combinators to write a simple translator from s-expressions to Factor quotations. The translation was quite simple - here are some examples of how it was supposed to work:

```
(set a 10)
  => [ "set" "a" 10 ]
(set square (lambda (n) (* n n)))
  => [ "set" "square" [ "lambda" [ "n" ] [ "*" "n" "n" ] ] ]
```

This looked like it should be fairly simple so I put together a quick implementation.

The parser combinators were recently rewritten so this code is likely to work best with the Darcs version of Factor. Instructions on how to get this going are at the Factor website. Basic instructions for an Intel Linux based machine would go something like:

```
git clone http://www.factorcode.org/
cd factor
make linux-x86-32
wget http://factorcode.org/images/latest/boot.x86.32.image
./factor -i=boot.x86.32.image
./factor
```

To run the examples from within Factor you will need to load the parser-combinators module and use it:

```
USE: parser-combinators
USE: lazy-lists
```

A parser combinator is a tuple that has specialised the ‘parse’ generic word. This word has stack effect (input parser – result). The input is a sequence of tokens (usually a string) and the result is what is known as a lazy list of successes.

A ‘list of sucesses’ is a list of all possible sucessful parse results. It being lazy, each result in the list is not computed (ie. the parse is not done) until that list element is requested. This is how parser combinators handle backtracking. If the first parse result in the list is not what is required, the second can be tried, etc.

Parsers can be written manually or existing parsers can be combined together using combinators. A number of existing parsers and combinators are provided in the ‘parser-combinators’ vocabulary to build simple parsers.

To start with the s-expression parser we can handle the ‘(’ and ‘)’ using the ‘token’ word. This word returns a parser that parses only the given string:

```
LAZY: 'lparens' ( -- parser )
  "(" token ;

LAZY: 'rparens' ( -- parser )
  ")" token ;
```

Notice these are defined with the ‘LAZY:’ word instead of ‘:’. This means the result of the word is a promise and needs to be forced to get the actual value. Although not strictly necessary with these simple parsers it is required for those that recursively call themselves so I generally always define parsers using it. Examples of usage:

```
"(" 'lparens' parse car . => T{ parse-result f "(" T{ slice f "(" 1 1 } }
```

The result of the parse is a lazy list. Returning the ‘car’ of that list shows the first parse result. The ‘parse-result’ tuple has two slots. The first slot is the parse tree returned by the parser. The second is the rest of the input string remaining to be parsed if any. Usually the latter is a slice for efficiency purposes. The parsers created with ‘token’ return the token itself as the parse tree.

The s-expression parser needs to parse numbers and identifiers. Numbers are digits repeated multiple times. A parser for digits could be:

```
LAZY: 'digit' ( -- parser )
  [ digit? ] satisfy ;
```

This uses the ‘satisfy’ parser generator word. Where ‘token’ parses an input string for a specified string, ‘satisfy’ will call a given quotation with the first character of the input string. If the quotation returns ‘true’ then the parse is succesful otherwise it fails. Examples of using this definition of ‘digit’ are:

```
"123" 'digit' parse car parse-result-parsed . => 49
```

The ‘parse-result-parsed’ word returns just the parse tree of the result from the ‘parse-result’ tuple. The result, ‘49’, is the character code of the successfully parsed digit. Ideally I want the actual number returned rather than the character code.

The ‘<@’ word applies a transformation to the parse tree of a parser. It converts an existing parser to one that does what the original parser did but has the transformation applied. ‘<@’ takes a quotation on the stack with stack effect (old-tree – new-tree). This quotation is called to perform the transformation. One way to remember the effect of ‘<@’ is to note that ‘<’ sign points to the parser being transformed. Here is ‘digit’ converted to return a number:

```
LAZY: 'digit' ( -- parser )
  [ digit? ] satisfy [ digit> ] <@ ;

"123" 'digit' parse car parse-result-parsed . => 1
```

‘digit>’ is a standard Factor word that coverts the character code of a digit into the digit itself. A number is one or more digits in a row. The <+> word takes a parser as input and returns a parser that parses one or more instances of the original. It has the same meaning as ‘+’ in regular expressions. This allows a ‘number’ parser word to be written as:

```
LAZY: 'number' ( -- parser )
  'digit' <+> ;

"123" 'number' parse car parse-result-parsed . => { 1 2 3 }
```

Notice here that the result is an array of the digits in the number. This is the parse tree that <+> generates - an array of the results of the original parser. By using ‘<@’ this can be converted into a numeric value using Factor’s ‘reduce’ word (basically a left fold):

```
LAZY: 'number' ( -- parser )
  'digit' <+> [ 0 [ swap 10 * + ] reduce ] <@ ;

"123" 'number' parse car parse-result-parsed . => 123
```

Interestingly ‘number’ actually returns more than one parse result since ”123” contains more than one occurrence of digits in a sequence:

```
"123" 'number' parse [ parse-result-parsed ] lmap [ . ] leach
  => 123
     12
     1
```

It returns the longest match first which is what we usually want. As the list is lazy if we don't request anything beyond the first match then the remaining parse results aren't actually computed.

Next on the list is identifiers. These are any sequences of characters that are not numbers, spaces, or parenthesis. The 'satisfy' word can be used for this, combined with <+>:

```
LAZY: 'identifier' ( -- parser )
  [
    [ blank? not ] keep
    [ digit? not ] keep
    [ CHAR: ( = not ] keep
    CHAR: ) = not
    and and and
  ] satisfy <+> [ >string ] <@ ;

"foo" 'identifier' parse car parse-result-parsed . => "foo"
```

The '<@' word is used to transform the result into a string otherwise we'd get an array of the character codes in the identifier as a result.

Often it is desired to parse either a number or an identifier. The 'atom' word does this:

```
LAZY: 'atom' ( -- parser )
  'identifier' 'number' <|> ;

"123" 'atom' parse car parse-result-parsed . => 123
"foo" 'atom' parse car parse-result-parsed . => "foo"
```

It uses '< | >', known as the choice word. It takes two parsers on the stack, and generates a parser which when run will try the first parser on the input string. If it fails it will then try the second parser. It returns the list of successes resulting from both parses.

A first cut at parsing a single expression like '(set a 10)' requires a sequencing parser combinator. This is '<&>'. It takes two parsers and returns a resulting parser which when run will run the first parser on the input string, and then run the second parser on the remaining unparsed portion of the input string of each result from the first parser. A simple expression parser might look like:

```
LAZY: 'expr1' ( -- parser )
  'lparens'
  'atom' <&>
  'rparens' <&> ;

"(123)" 'expr1' parse car parse-result-parsed . =>
  { { "(" 123 } ")" }
```

This results in a very ugly parse tree. The '<&>' word returns a parse tree which is an array of the two parsers it uses. Since we have two '<&>' calls we get nested results. This can be fixed using two variants of '<&>'. They are '<&' and '&>'. These words do the same as '<&>' but don't put the result of one of the parsers in the parse tree. The '>' or '<' point to the parser that will have the result returned. Here's a new version that has a nicer parse tree using these words:

```
LAZY: 'expr2' ( -- parser )
  'lparens'
  'atom' &>
  'rparens' <& ;

"(123)" 'expr2' parse car parse-result-parsed . => 123
```

Notice the variants of '<&>' used both select the result of the 'atom' parser to be included in the parse tree and not the results of the parenthesis parsers.

There is still a problem with the expression parser. It doesn't handle more than one 'atom' in the expression. Similar to '<+>' there is '<*>' which takes a parser and returns a new parser which when called parses zero or more instances of the original parser. The parse tree for the result of '<*>' is an array of the results of the original parser:

```
LAZY: 'expr3' ( -- parser )
  'lparens'
  'atom' sp <*> &>
  'rparens' <& ;

"(set a 123)" 'expr3' parse car parse-result-parsed . =>
  { "set" "a" 123 }
```

This has one other change in that it needs to handle white space between atoms. The 'sp' word takes a parser (in this case 'atom') and returns a parser that first removes all white space from the start of the input string and then calls the original parser. The effect is to produce a parser that ignores white space.

This gets close to what out test case requires but it returns an array not a quotation. Using '<@' fixes this:

```
LAZY: 'expr4' ( -- parser )
  'lparens'
  'atom' sp <*> &>
  'rparens' <& [ >quotation ] <@ ;

"(set a 123)" 'expr4' parse car parse-result-parsed . =>
  [ "set" "a" 123 ]
```

Unfortunately this fails on our second test case which requires handling nested expressions like '(+ 1 (+ 2 3) 4)'. By recursively calling the 'expr' word this is easy to add:

```
LAZY: 'expr5' ( -- parser )
  'lparens'
  'atom' 'expr5' <|> sp <*> &>
  'rparens' <& [ >quotation ] <@ ;

"(+ 10 (+ 20 30))" 'expr5' parse car parse-result-parsed . =>
  [ "+" 10 [ "+" 20 30 ] ]
```

This change was as simple as replacing the 'atom' parser used between the two parenthesis with a parser which checks for an atom or an expression. Note that this recursively calls itself, which is safe from infinite recursion due to the lazy evaluation. It doesn't actually get evaluated unless the 'atom' test fails first. One final change is to allow for only atoms as well as expressions and then our simple test cases work and it is completed:

```
LAZY: 'expr' ( -- parser )
  'atom'
  'lparens'
  'atom' 'expr' <|> sp <*> &>
  'rparens' <& [ >quotation ] <@ <|> ;

"(set square (lambda (n) (* n n)))" 'expr' parse car parse-result-parsed . =
 [ "set" "square" [ "lambda" [ "n" ] [ "*" "n" "n" ] ] ]

"123" 'expr' parse car parse-result-parsed . =>
  123

"hello" 'expr' parse car parse-result-parsed . =>
  "hello"
```

These examples should work using the parser combinators code in the current Factor darcs repository and in the upcoming 0.85 release. I'm currently writing more documentation for the parser combinators and it will be accessable using the standard Factor help system. The source for this example can be found here.

Chapter 12

Compilers and Interpreters

Writing compilers and interpreters in Factor is quite easy. By writing the grammar using parser combinators, and having those combinators produce an abstract syntax tree (AST), it's easy to write a simple compiler or interpreter for the tree. I've already covered writing simple parser combinators so I'll concentrate on processing the AST for a simple language in this post. A later post might generate a parser for it or it could be an 'exercise for the reader'.

For an example to start with I took the evaluator defined in this Lambda the Ultimate posting and converted it to Factor. For the interpreter I tried two different approaches to compare them. The first was to use pattern matching. For the second I use generic functions and Factor's OO system.

For the compiler I show how to compile to Factor code, which can then be interpreted or compiled by Factor itself, and I also compile to Javascript. This javascript can be run in a standalone Javascript interpreter like Rhino, or in a web browser.

To more easily re-use the AST across the two examples I added the ability to pattern match on tuples to the 'match' contrib library. This can be obtained from the standard Factor git repository:

```
git clone git://factorcode.org/git/factor.git
```

The factor 'latest' boot images can be used to bootstrap from this repository. The instructions on how to do this are on factorcode.org. For an Intel x86 linux machine the steps would be:

```
git clone git://factorcode.org/git/factor.git
cd factor
make linux-x86-32
wget http://factorcode.org/images/latest/boot.x86.32.image
./factor -i=boot.x86.32.image
```

The examples here should also work with the released Factor 0.85 as long as you replace 'contrib/match/match.factor' with the version from my repository. To load the 'match' library in Factor use:

```
USE: match
```

In the Factor code snippets below I use '=¿' to show the result of running Factor words. You shouldn't actually type that. Instead the result will be shown on the Factor stack or you can use '.' to print it.

The code for the examples in this post can be downloaded from interpreter.factor. If you copy it into the 'contrib' directory of your Factor installation you can load it with:

```
"contrib/interpreter" require
```

The evaluator that is being implemented has the following description from the Lambda the Ultimate posting:

```
data Term a where
    Lit  :: Int -> Term Int
    Inc  :: Term Int -> Term Int
    IsZ  :: Term Int -> Term Bool
    If   :: Term Bool -> Term a -> Term a -> Term a
    Pair :: Term a -> Term b -> Term (a,b)
    Fst  :: Term (a,b) -> Term a
    Snd  :: Term (a,b) -> Term b

eval :: Term a -> a
eval (Lit i)    = i
eval (Inc t)    = eval t + 1
eval (IsZ t)    = eval t == 0
eval (If b t e) = if eval b then eval t else eval e
eval (Pair a b) = (eval a, eval b)
eval (Fst t)    = fst (eval t)
eval (Snd t)    = snd (eval t)
```

Basically we have literal values, which in this example are integers. These can be incremented and tested to see if they are zero. A 'pair' can be created and the first and second elements obtained from the pair. An 'if' expression is available for testing an expression and evaluating a true or false clause. These are represented using Factor tuples:

```
TUPLE: lit i ;
TUPLE: inc t ;
```

```
TUPLE: isz t ;
TUPLE: iff b t e ;
TUPLE: pair a b ;
TUPLE: fst t ;
TUPLE: snd t ;
```

I used the same names for the types and elements as the example but changed ‘if’ to ‘iff’ to prevent clashing with Factor’s standard ‘if’ word. An example tree can be created that is the increment of the number 5 with:

```
5 <lit> <inc>
```

When evaluated this should produce the result ‘6’.

The interpreter implementation that uses pattern matching uses ‘match-cond’. This requires match variables to be set up which can be used to destructure sequences and tuples. For example:

```
5 <lit> T{ lit f ?t } match [ ?t ] bind => 5
```

‘T lit f 123 ’ is the Factor syntax for a tuple. It creates a tuple at parse time and the actual object is stored in the word. This is different from ’123 ¡lit¿’ which will produce the literal object at runtime. ‘?t’ is a matching variable and the ‘match’ and ‘match-cond’ words use these to destructure matching elements of tuples and sequences. The match variable can then be used to get the actual value obtained. For more on the Factor pattern matching system you can read the documentation from within Factor with:

```
\ match help
```

The Factor implementation of the interpreter using pattern matching is:

```
: eval1 ( a -- a )
  {
    { T{ lit  f ?i }       [ ?i ] }
    { T{ inc  f ?t }       [ ?t eval1 1+ ] }
    { T{ isz  f ?t }       [ ?t eval1 zero? ] }
    { T{ iff  f ?b ?t ?e } [ ?b eval1 [ ?t ] [ ?e ] if eval1 ] }
    { T{ pair f ?a ?b }    [ ?a eval1 ?b eval1 2array ] }
    { T{ fst  f ?t }       [ ?t eval1 first ] }
    { T{ snd  f ?t }       [ ?t eval1 second ] }
  } match-cond ;
```

It is pretty much a direct translation of the Haskell example. An example of using it:

```
5 <lit> <inc> eval1 => 6
```

The Lambda the Ultimate post gave a usage example:

```
stuff = Pair (If (IsZ (Inc (Inc (Lit 5))))
                 (Lit 6)
                 (Lit 7))
             (Fst (Snd (Pair (Lit 42)
                             (Pair (Lit 8) (Lit 9))))))
```

Translated to Factor it is:

```
: driver ( -- v )
  5 <lit> <inc> <inc> <isz>
    6 <lit>
    7 <lit>
  <iff>
  42 <lit> 8 <lit> 9 <lit> <pair> <pair> <snd> <fst> <pair> ;
```

Running the ‘eval1’ word on this produces the answer, 7 8 :

```
driver eval1 => { 7 8 }
```

Although ‘eval1’ is quite short and easy to understand it has a disadvantage in that to extend it with new AST types you need to modify the ‘eval1’ word. Using the Factor OO generic word system you extend the interpreter without having to modify existing code. Here’s a generic word approach to the interpreter:

```
GENERIC: eval2 ( a -- a )

M: lit  eval2 ( a -- a ) lit-i ;
M: inc  eval2 ( a -- a ) inc-t eval2 1+ ;
M: isz  eval2 ( a -- a ) isz-t eval2 zero? ;
M: iff  eval2 ( a -- a ) dup iff-b eval2 [ iff-t ] [ iff-e ] if eval2 ;
M: pair eval2 ( a -- a ) dup pair-a eval2 swap pair-b eval2 2array ;
M: fst  eval2 ( a -- a ) fst-t eval2 first ;
M: snd  eval2 ( a -- a ) snd-t eval2 second ;
```

The ‘M:’ word is used to define a method for a generic word. This is similar to how generic functions and methods work in CLOS and Dylan. When a generic word is called the actual method invoked is based on the topmost item of the stack. The first symbol past the ‘M:’ is the type of that object that that method is for. The second is the name of the generic word the method will be added too. The rest of the definitions should be reasonably clear - they break apart the tuples and return results or call ‘eval2’ recursively. ‘eval2’ produces the same result as ‘eval1’:

```
driver eval2 => { 7 8 }
```

Which approach to use is really personal preference. Of the two, ‘eval2’ is the most efficient. This is because methods can be compiled in Factor whereas the current implementation of ‘match-cond’ cannot be. So ‘eval1’ will run in the Factor interpreter whereas ‘eval2’ will be compiled to native code. This can be seen by trying to compile the examples and running a timed test:

```
\ eval1 compile => "The word eval1 does not have a stack effect"
\ eval2 compile
\ eval1 compiled? => f
\ eval2 compiled? => t
[ 1000 [ driver eval1 ] times ] time => 487ms run / 5 ms GC time
[ 1000 [ driver eval2 ] times ] time => 3ms run / 0 ms GC time
```

There is obviously room for improvement in the ‘match’ routines! I can say that because I wrote them :)

A compiler follows the same structure as the interpreter but instead of computing the result we generate code. This code is then later fed to the compiler. For the compiler examples I’m only going to show the pattern matching based example. The generic word implementation is very similar and would follow the relevant interpreter implementation.

Factor provides a ‘make’ word that can be used for dynamically appending to a new sequence. From within a ‘make’ scope you can use ‘,’ to append an element and ‘%’ to splice in a sequence to the constructed sequence. ‘make’ can be used for constructing arrays, strings, quotations, etc. The type of the constructed sequence is identified by an exemplar sequence passed to ‘make’. For example:

```
[ 1 , 2 , { 3 4 } % { 5 } , ] { } make => { 1 2 3 4 { 5 } }
```

The top level ‘compile’ word will set up a ‘make’ scope for the AST compile routines to append to. Here’s the compiler to Factor:

```
: (compile1) ( a -- )
  {
    { T{ lit  f ?i }       [ ?i , ] }
    { T{ inc  f ?t }       [ ?t (compile1) \ 1+ , ] }
    { T{ isz  f ?t }       [ ?t (compile1) \ zero? , ] }
    { T{ iff  f ?b ?t ?e } [ ?b (compile1)
                             [ ?t (compile1) ] [ ] make ,
                             [ ?e (compile1) ] [ ] make ,
                             \ if , ] }
    { T{ pair f ?a ?b }    [ ?a (compile1) ?b (compile1) \ 2array , ] }
```

```
    { T{ fst  f ?t }       [ ?t (compile1) \ first , ] }
    { T{ snd  f ?t }       [ ?t (compile1) \ second , ] }
  } match-cond ;

: compile1 ( a -- quot )
  [ (compile1) ] [ ] make ;
```

As you can see it is very similar to the interpreter. Instead of returning the result of the AST evaluation '(compile1)' appends the equivalent Factor code to the constructed quotation created in the 'compile1' word.

The implementation for handling 'lit' just appends the numeric value directly. For 'inc' it calls '(compile1)' on the term to be incremented so that gets appended to the quotation. It then appends the Factor '1+' word which will be used to increment it. The 'ís an escape mechanism to tell Factor to store the word itself rather than call it.

The main complication is in the implementation of 'iff'. This word must delay the computation of the two terms of the 'iff' statement. To to this it creates Factor quotations with 'make' and then recursively calls '(compile1)' for the terms. The results of the compilation for this recursive call will be stored in the most recently created quotation rather than the one in the top level 'compile1' word. That is, nested 'make' calls are dynamically scoped.

An example of usage of the compiler:

```
5 <lit> <inc> compile1 => [ 5 1 + ]
[ 5 1 + ] call => 6
```

The 'driver' AST shown previously compiles to Factor correctly:

```
driver compile1 => [
  5 1+ 1+ zero? [
    6
  ] [
    7
  ] if 42 8 9 2array 2array second first 2array
]
```

This can be compiled to native code using Factor or run in the Factor interpreter:

```
driver compile1 compile-quot execute => { 7 8 }
driver compile1 call => { 7 8 }
```

The final example is a compiler to Javascript. Again it follows the same basic outline of the interpreter. Instead of generating a Factor quotation it uses 'make' to generate a string:

```
: (compile2) ( a -- )
  {
    { T{ lit  f ?i }       [ ?i number>string % ] }
    { T{ inc  f ?t }       [ ?t (compile2) "+1" % ] }
    { T{ isz  f ?t }       [ ?t (compile2) "== 0" % ] }
    { T{ iff  f ?b ?t ?e } [
        "function() {if(" %
        ?b (compile2)
        ") {return " %
        ?t (compile2)
        "} else { return " %
        ?e (compile2)
        "}}()" %
      ]
    }
    { T{ pair f ?a ?b }    [ "{ first:" % ?a (compile2) ",second:" % ?b (compile2) "
    { T{ fst  f ?t }       [ ?t (compile2) ".first" % ] }
    { T{ snd  f ?t }       [ ?t (compile2) ".second" % ] }
  } match-cond ;

: compile2 ( a -- quot )
  [ "(" % (compile2) ")" % ] "" make ;
```

The main complication with this compiler was handling ‘if’. The Javascript ‘if’ has no result so cannot be assigned immediately. So I wrap the ‘if’ in a function which is called immediately. The two terms of the ‘if’ use ‘return’ to return the result from the function. A simple example:

```
5 <lit> <inc> compile2 => "(5+1)"
```

And ‘driver’ also compiles successfully:

```
driver compile2 =>
"({
    first:function() {
      if(5+1+1== 0) {
        return 6
      } else {
        return 7
      }
    }(),
    second: {
      first:42,
      second: { first:8,second:9 }
    }.second.first
})"
```

This result evaluates successfully in a browser. You can test it by clicking here.

Chapter 13

Web Applications

There are a number of different ways of writing web applications in Factor but for this approach I'm using the furnace framework.

The first step is to start the web server. This lives in the vocab `http.server`:

```
USE: http.server
[ 8888 httpd ] in-thread
```

This will start an instance of the server on port 8888 in another thread, to allow us to continue to enter commands in the listener.

By default web applications are accessed on the URL path `/responder/name`, where `name` is the name of the web application.

Accessing the web application path runs an 'action'. An action produces HTML output which gets sent back to the client browser. A web application has a default 'action' that gets run (the equivalent of an index.html), and can have other actions that are specified in the URL. Some examples:

```
http://localhost:8888/responder/foo
    Runs the default action for the 'foo' web application
http://localhost:8888/responder/foo/doit
    Runs the 'doit' action
http://localhost:8888/responder/foo/hello?name=chris
    Runs the 'hello' action giving the argument 'name' with the value 'chris'
```

The syntax for furnace URL's is therefore `http://servername:port/responder/[webappname]/[action]?[a`

Furnace web application must exist under the `webapps` vocabulary. So accessing `/responder/foo` will look for furnace details in the vocabulary `webapps.foo`.

A furnace web application is registered with the http server using the `web-app` word. It takes three arguments on the stack.

Vocabulary: `furnace`
Word: `web-app ( name default path -- )`

The `name` is the vocabulary name of the web application with out the `webapps` prefix. `default` is the name of the action that gets run when the web application URL is accessed. `path` is the location of any template files the web application uses.

An action is a word that outputs data to be sent to the browser. It can be as simple as:

```
: doit ( -- ) serving-text "I am here" print ;
```

The word must be registered as an action:

```
\ doit { } define-action
```

Now accessing the URL for the web application with `doit` at the end of the path will result in 'I am here' being sent to the browser. Note the `serving-text` call. That outputs the headers for the mime type and the standard HTTP response. There is also a `serving-html`, or you could write the headers manually.

Actions can take arguments. These are placed on the stack for the word that is called:

```
: hello ( name -- ) serving-text "Hello " write print ;
\ hello { { "hello" } } define-action
```

So the complete code for the simplest of web applications is:

```
USE: http.server
[ 8888 httpd ] in-thread
IN: webapps.test
USE: furnace

: index serving-text "We're alive!" print ;
\ index { } define-action

: hello ( name -- ) serving-text "Hello " write print ;
\ hello { { "name" } } define-action

"test" "index" "." web-app
```

Accessing `http://localhost:8888/responder/test` will run the 'index' action. This is what we passed as the 'default' parameter on the stack to the 'web-app' word. Accessing `http://localhost:8888/responder/test/hello?name=chris` will run the 'hello' action.

There is also the facility to have template files, very much like JSP. The 'path' parameter to `web-app` defines the location of these. Inside your action word you can call 'render-template' to run the template and have it sent to the browser:

```
: runme ( -- ) f "page" "Title" render-template ;
\ runme { } define-action
```

This will load the ‘page.furnace’ file in the path given to ‘web-app’. It should contain standard HTML with embedded Factor code inside <% and %> tags. It will be run and sent to the client. The ‘f’ passed in this example can be an instance of a tuple (an object) and the template can access the slots of that instance to display data, etc.

There is quite a bit more that can be done. There is a continuation based workflow system, validators for actions, etc. There is also much more that needs to be done. handling sessions, cookies, etc. Hopefully this post gives a quick introduction and allows you to get started.

Chapter 14

Continuation Based Web Apps

14.1 Overview

The ‘cont-responder’ library is a continuation based web server for writing web applications in Factor. Each ‘web application’ is a standard Factor httpd responder.

This document outlines how to write simple web applications using ‘cont-responder’ by showing examples. It does not attempt to go into the technical details of continuation based web applications or how it is implemented in Factor. Instead it uses a series of examples that can be immediately tried at the Factor prompt to get a feel for how things work.

14.2 Getting Started

To get started you will first need to use the ‘cont-responder’ vocabulary:

```
USE: cont-responder
```

The responders that you will be writing will require an instance of the httpd server to be running. It will be run in a background thread to enable the interactive development of the applications. The following is a simple function to start the server on port 8888:

```
USE: httpd
USE: threads
: start-httpd [ 8888 httpd ] in-thread  ;
start-httpd
```

14.3 Responders

A ‘responder’ is a word that is registered with the httpd server that gets run when the client accesses a particular URL. When run that word has ‘standard output’ bound in such a way that all output goes to the clients web browser.

In the ‘cont-responder’ system there are two words used to set output to go to the web browser and display a page. They are ‘show’ and ‘show-final’. Think of them as ‘show a page to the client’. ‘show’ and ‘show-final’ both take a single item on the stack and that is a ‘page generation’ quotation.

A ‘page generation’ quotation is a quotation which when called will output HTML to stdout. In the httpd system, stdout is bound to the socket connection to the clients web browser.

The ‘page generation’ quotation passed to ‘show’ should have stack effect (string –) while that for ‘show-final’ has stack effect (–). The two words have very similar uses.

The big difference is with ‘show’. It provides an URL to the page generation quotation that when requested will proceed with execution immediately following the ‘show’, and any POST request data will be on the stack. With ‘show-final’ no URL is passed so it is not possible to ‘resume’ computation. This is explained more fully later.

14.4 Hello World 1

A simple ‘hello world’ responder would be:

```
: hello-world1 ( -- )
  [
    "<html><head><title>Hello World</title></head>" write
    "<body>Hello World!</body></html>" write
  ] show-final ;
```

When installed this will show a single page which is simple HTML to display ‘Hello World¡.

The responder is installed using:

```
"helloworld1" [ hello-world1 ] install-cont-responder
```

The ‘install-cont-responder’ word has stack effect (name quot –). It installs a responder with the given name.

When the URL for that responder is accessed the ‘quot’ quotation is run. In this case it is ‘hello-world1’ which displays the single HTML page described previously.

Accessing the above responder from a web browser is via an URL like:

```
http://localhost:8888/responder/helloworld1
```

This should display an HTML page showing “Hello World!”.

14.5 HTML Generation

Generating HTML by writing strings containing HTML can be a bit of a chore. Especially when the content is dynamic requiring concatenation of many pieces of data.

The ‘cont-responder’ system uses ‘html’, a library that allows writing HTML looking output directly in factor. This system, developed for ‘cont-responder’, has recently been made part of the standard ‘html’ library of Factor.

‘html’ basically allows you to write HTML-like output in a factor word and it will be output as correct HTML. It can be tested at the console very easily:

```
USE: html
<p> "This is a paragraph" write </p>
  => <p>This is a paragraph</p>
```

You can write open and close tags like ordinary HTML and anything sent to standard output in between the tags will be enclosed in the specified tags. Attributes can also be used:

```
<p "text-align: center" =style p> "More text" write </p>
  => <p style='text-align: center'>More text</p>
```

The attribute must be seperated from the value of that attribute via whitespace. If you are using attributes the tag must be closed with a ‘[tagname]>’ where the [tagname] is the name of the tag used. See the ‘<p p>’ example above.

You can use any factor code at any point:

```
"text-align: " "red"
<p 2dup cat2 =style p>
  "Using style " write swap write write
</p>
  => <p style='text-align: red'>Using style text-align: red</p>
```

Tags that are not normally closed are written using XML style closed tag (ie. with a trailing slash):

```
"One" write <br/> "Two" write <br/> <input "text" =type input/>
  => One<br>Two<br><input type='text'>
```

14.6 Hello World 2

Using the HTML generation library makes writing responders more readable. Here is the hello world example perviously using this system:

```
: hello-world2 ( -- )
  [
    <html>
      <head> <title> "Hello World" write </title> </head>
      <body> "Hello World!" write </body>
    </html>
  ] show-final ;
```

Install it using:

```
"helloworld2" [ hello-world2 ] install-cont-responder
```

14.7 Dynamic Data

Adding dynamic data to the page is relatively easy. This example pulls information from the 'room' word which returns memory details about the running Factor system. It also uses 'room.' which outputs these details to standard output and this is wrapped in a <pre> tag so it is formatted correctly.

```
: memory-stats1 ( -- )
  [
    <html>
      <head> <title> "Memory Statistics" write </title> </head>
      <body>
        <table "1" =border table>
          <tr>
            <td> "Total Data Memory" write </td>
            <td> room unparse write </td>
          </tr>
          <tr>
            <td> "Free Data Memory" write </td>
            <td> unparse write </td>
          </tr>
          <tr>
            <td> "Total Code Memory" write </td>
            <td> unparse write </td>
          </tr>
          <tr>
```

```
              <td> "Free Code Memory" write </td>
              <td> unparse write </td>
            </tr>
  </table>
</body>
        <pre> room. </pre>
      </html>
    ] show-final ;

  "memorystats1" [ memory-stats1 ] install-cont-responder
```

Accessing this page will show a table with the current memory statistics. Hitting refresh will update the page with the latest information.

The HTML output can be refactored into different words. For example:

```
  : memory-stats-table ( free total -- )
    #! Output a table containing the given statistics.
    <table "1" =border table>
      <tr>
        <td> "Total Data Memory" write </td>
        <td> unparse write </td>
      </tr>
      <tr>
        <td> "Free Data Memory" write </td>
        <td> unparse write </td>
      </tr>
    </table> ;

  : memory-stats2 ( -- )
    [
      <html>
        <head> <title> "Memory Statistics 2" write </title> </head>
        <body> room memory-stats-table 2drop </body>
      </html>
    ] show-final ;

  "memorystats2" [ memory-stats2 ] install-cont-responder
```

14.8 Some simple flow

The big advantage with continuation based web servers is being able to write a web application in a standard procedural flow and have it correctly served up in the HTTP request/response model.

This example demonstates a flow of three pages. Clicking an URL on the first page displays the second. Clicking an URL on the second displays the third.

When a 'show' call is executed the page generated by the quotation is sent to the client. The computation of the responder is then 'suspended'. When the client accesses a special URL, computation is resumed at the point of the end of the 'show' call. In this way procedural flow is maintained.

This brings us to the 'URL' stack item that is available to the 'page generation' quotation passed to 'show'. This URL is a string that contains an URL that can be embedded in the page. When the user access that URL, computation is resumed from the point of the end of the 'show' call as described above:

```
: flow-example1 ( -- )
  [
    <html>
      <head> <title> "Flow Example 1" write </title> </head>
      <body>
        <p> "Page 1" write </p>
        <p> <a =href a> "Press to continue" write </a> </p>
      </body>
    </html>
  ] show drop
  [
    <html>
      <head> <title> "Flow Example 1" write </title> </head>
      <body>
        <p> "Page 2" write </p>
        <p> <a =href a> "Press to continue" write </a> </p>
      </body>
    </html>
  ] show drop
  [
    <html>
      <head> <title> "Flow Example 1" write </title> </head>
      <body>
        <p> "Page 3" write </p>
      </body>
    </html>
  ] show-final ;

"flowexample1" [ flow-example1 ] install-cont-responder
```

The 'flow-example1' word contains two 'show' calls in a row, followed by a 'show-final'. The 'show' calls display simple pages with an anchor link to the URL received on the stack. This URL when accessed resumes the computation. The final page

doesn't require resumption of the computation so 'show-final' is used. We could have used 'show' and dropped the URL passed to the quotation and the result following the 'show' but using 'show-final' is more efficient.

When you display this example in the browser you'll be able to click the URL to navigate. You can use the back button to retry the URL's, you can clone the browser window and navigate them independantly, etc.

The similarity of the functions above shows that some refactoring would be useful. The pages are almost exactly the same so we seperate this into a seperate word:

```
: show-flow-page ( n bool -- )
#! Show a page in the flow, using 'n' as the page number
#! to display. If 'bool' is true display a link to the
#! next page.
[ ( n bool url -- )
  <html>
    <head> <title> "Flow Example 1" write </title> </head>
    <body>
      <p> "Page " write rot unparse write </p>
      swap [
        <p> <a =href a> "Press to continue" write </a> </p>
      ] [
        drop
      ] ifte
    </body>
  </html>
] show 3drop ;

: flow-example2 ( n -- )
#! Display the given number of pages in a row.
dup 1 - [ dup 1 + t show-flow-page ] repeat
f show-flow-page ;

"flowexample2" [ 5 flow-example2 ] install-cont-responder
```

In this example the 'show-flow-page' pulls the page number off the stack. It also gets whether or not to display the link to the next page.

Notice that after the show that a '3drop' is done whereas previously we've only done a single 'drop'. This is due to a side effect of 'show' using continuations.

After the 'show' call returns there will be one item on the stack (which we've been dropping and will explain later what it is). The stack will also be set as it was before the show call. So in this case the 'n' and 'bool' remain on the stack even though they were removed during the page generation quotation. This is because we resumed the continuation which, when captured, had those items on the stack. The general rule of thumb is you will need to account for items on the stack before the show call.

This example also demonstrates using the 'repeat' combinator to sequence the page shows. Any Factor code can be called and the continuation based system will sequentially display each page. The back button, browser window cloning, etc will all continue to work.

You'll notice the URL's in the browser have an 'id' query parameter with a sequence of characters as its value. This is the 'continuation identifier' which is like a session id except that it identifies not just the data you have stored but your location within the responder as well.

14.9 Forms and POST data

The web pages we've generated so far don't accept input from the user. I've mentioned previously that 'show' returns a value on the stack and we've been dropping it in our examples.

The value returned is a namespace containing the field names and values of any POST data in the request. If no POST data exists then it is the boolean value 'f'.

To process input from the user just put a form in the HTML with a method of 'POST' and an action set to the URL passed in to the page generation quotation. The show call will then return a namespace containing this data. Here is a simple example:

```
: accept-users-name ( -- name )
  #! Display an HTML requesting the users name. Push
  #! the name the user input on the stack..
  [
    <html>
      <head> <title> "Please enter your name" write </title> </head>
      <body>
        <form =action "post" =method form>
          <p>
            "Please enter your name:" write
            <input "text" =type "20" =size "username" =name input/>
            <input "submit" =type "Ok" =value input/>
          </p>
        </form>
      </body>
    </html>
  ] show [
    "username" get
  ] bind ;

: post-example1 ( -- )
  [
```

```
      <html>
        <head> <title> "Hello!" write </title> </head>
        <body>
          <p> accept-users-name write ", Good to see you!" write </p>
        </body>
      </html>
    ] show-final  ;

"post-example1" [ post-example1 ] install-cont-responder
```

The 'accept-users-name' word displays an HTML form allowing input of the name. When that form is submitted the namespace containing the data is returned by 'show'. We bind to it and retrieve the 'username' field. The name used here should be the same name used when creating the field in the HTML.

'post-example1' then does something a bit tricky. Instead of first calling 'accept-users-name' to push the name on the stack and then displaying the resulting page we call 'accept-users-name' from within the page itself when we actually need it. The magic of the continuation system causes the 'accept-users-name' to be called when needed displaying that page first. It is certainly possible to do it the other way though:

```
: post-example2 ( -- )
  accept-users-name
  [ ( name url -- )
    <html>
      <head> <title> "Hello!" write </title> </head>
      <body>
        <p> write ", Good to see you!" write </p>
      </body>
    </html>
  ] show-final ;

"post-example2" [ post-example2 ] install-cont-responder
```

14.10 Associating URL's with words

A web page can contain URL's that when clicked perform some action. This may be to display other pages, or to affect some server state.

The 'cont-responder' system enables an URL to be associated with any Factor quotation. This quotation will be run when the URL is clicked. When that quotation exits control is returned to the page that contained the call.

The word that enables this is 'quot-href'. It takes two items on the stack. One is the text to display for the link. The other is the quotation to run when the link is clicked. This quotation should have stack effect (–).

This example displays a number which can be incremented or decremented.

```
0 "counter" set

: counter-example1 ( - )
  #! Display a global counter which can be incremented or decremented
  #! using anchors.
  [
    <html>
      <head>
        <title> "Counter: " write "counter" get unparse dup write  </title>
      </head>
      <body>
        <h2> "Counter: " write write </h2>
        <p>  "++" [ "counter" get 1 + "counter" set ] quot-href
             "--" [ "counter" get 1 - "counter" set ] quot-href
        </p>
      </body>
    </html>
  ] show-final ;

  "counter-example1" [ counter-example1 ] install-cont-responder
```

Accessing this example from the web browser will display a count of zero. Clicking '++' or '–' will increment or decrement the count respectively. This is done by calling a quotation that either increments or decrements the count when the URL's are clicked.

Because the count is 'global' in this example, if you clone the browser window with the count set to a specific value and increment it, and then refresh the original browser window you will see the most recent incremented state. This gives you 'shopping cart' like state whereby using the back button or cloning windows shows a view of a single global value that can be modified by all browser instances. ie. The state is not backtracked when the back button is used.

You'll notice that when you visit the root URL for the responder that the count is reset back to zero. This is because when the responder was installed the value of zero was in the namespace stack. This stack is copied when the responder is installed resulting in initial accesses to the URL having the starting value. This gives you 'server side session data' for free.

14.11 Local State

You can also have a counter value with 'local' state. That is, cloning the browser window will give you a new independant state value that can be incremented. Going to the original browser window and refreshing will show the original value which can

be incremented or decremented seperately from that value in the cloned window. With this type of state, using the back button results in 'backtracking' the state value.

A way to get 'local' state is to store values on the stack itself rather than a namespace:

```
: counter-example2 ( count - )
  [ ( count URL -- )
    <html>
      <head>
        <title> "Counter: " write dup unparse write  </title>
      </head>
      <body>
        <h2> "Counter: " write dup unparse write </h2>
        <p>  "++" over [ 1 + counter-example2 ] cons quot-href
             "--" swap [ 1 - counter-example2 ] cons quot-href
        </p>
      </body>
    </html>
  ] show-final ;

  "counter-example2" [ 0 counter-example2 ] install-cont-responder
```

This example works by taking the value of the counter and consing it to a code quotation that will increment or decrement it then call the responder again. So if the counter value is '5' the two 'quot-href' calls become the equivalent of:

```
  "++" [ 5 1 + counter-example2 ] cons quot-href
  "--" [ 5 1 - counter-example2 ] cons quot-href
```

Because it calls itself with the new count value the state is remembered for that page only. This means that each page has an independant count value. You can clone or use the back button and all browser windows have an independant value.

14.12 Calling 'Subroutines'

Being able to call other page display functions from 'quot-href' gives you subroutine like functionality in your web pages. A simple menu that displays a sequence of pages and returns back to the main page is very easy:

```
  : show-page ( n -- )
  #! Show a page in the flow, using 'n' as the page number
  #! to display.
  [ ( n url -- )
```

```
    <html>
      <head> <title> "Page " write over unparse write </title> </head>
      <body>
        <p> "Page " write swap unparse write </p>
        <p> <a =href a> "Press to continue" write </a> </p>
      </body>
    </html>
  ] show 2drop ;

  : show-some-pages ( n -- )
  #! Display the given number of pages in a row.
  [ dup 1 + show-page ] repeat ;

  : subroutine-example1 ( -- )
    [
      <html>
        <head> <title> "Subroutine Example 1" write </title> </head>
        <body>
          <p> "Please select:" write
              <ol>
                <li> "Flow1" [ 1 show-some-pages ] quot-href </li>
                <li> "Flow2" [ 2 show-some-pages ] quot-href </li>
                <li> "Flow3" [ 3 show-some-pages ] quot-href </li>
              </ol>
          </p>
        </body>
      </html>
    ] show-final ;

  "subroutine-example1" [ subroutine-example1 ] install-cont-responder
```

Each item in the ordered list is an anchor. When pressed they will call a quotation that displays a certain number of pages in a row. When that quotation finishes via dropping off the end the main menu page is displayed again.

14.13 Simple Testing

Sometimes it is useful to test the responder words from the console instead of accessing it via a web browser. This enables you to step through or quickly check to see if a word is generating HTML correctly.

Because the responders require some state associated with them to keep track of continuation id's and other things you can't usually just run them and expect them to work. The 'show' call for example will fail as it expects some continuations to in the continuation table for that responder.

The ‘cont-testing.factor’ file (in the contrib/cont-responder directory) contains some simple words that maintains this state for you in such a way that you can test the words from the console:

```
"/contrib/cont-testing/load.factor" run-resource
```

For this example we’ll call the ‘subroutine-example1’ responder from above. First we need to put a ‘testing state’ object on the stack. All the testing functions expect this on the stack and return it after they have been called. We then put a quotation on the stack which calls the code we want to test and call the ‘test-cont-function’ word:

```
<cont-test-state> [ subroutine-example1 ] test-cont-function
 =>
HTTP/1.1 302 Document Moved
Location: ?id=8209741119458310
Content-Length: 0
Content-Type: text/plain
```

The first request is often a ‘Document Moved’ as above. This is because by default the ‘cont-responder’ system does the ‘Post-Refresh-Get’ pattern which results in a redirect after each request. This can be disabled but we’ll work through the example with it enabled.

We can see the continuation id where we are ‘moved’ to in the ‘Location’ header. To access this we use the ‘test-cont-click’ function. Think of this as manually clicking the URL. ‘test-cont-click’ has stack effect (state url post – state). ‘post’ is a hashtable of post data to pass along with the request. We use ‘f’ here because we have no post data. Remember that our previous ‘test-cont-function’ call left the state on the stack:

```
"8209741119458310" f test-cont-click
 =>
HTTP/1.0 200 Document follows
Content-Type: text/html
<html><head><title>Subroutine Example 1</title></head>
      <body><p>Please select:
               <ol><li><a href='?id=7687398605200513'>Flow1</a></li>
                   <li><a href='?id=7856272029924613'>Flow2</a></li>
                   <li><a href='?id=4909116160485714'>Flow3</a></li>
               </ol>
            </p>
      </body>
 </html>
```

We can continue to drill down using ‘test-cont-click’ using the URL’s above to see the HTML for each ‘click’.

Here’s an example using post data. We’ll test the ‘post-example1’ word written previously:

```
<cont-test-state> [ post-example1 ] test-cont-function
  =>
HTTP/1.1 302 Document Moved
Location: ?id=5829759941409535
Content-Length: 0
Content-Type: text/plain
```

Again we skip past the forward:

```
"5829759941409535" f test-cont-click
  =>
HTTP/1.0 200 Document follows
Content-Type: text/html

<html><head><title>Please enter your name</title></head>
      <body>
        <form action='?id=5456539333180428' method='post'>
          <p>Please enter your name:
             <input type='text'size='20'name='username'>
             <input type='submit'value='Ok'>
          </p>
        </form>
      </body>
</html>
```

Now we submit the post data along to the 'action' url:

```
"5456539333180428" [ [[ "username" "Chris" ]] ] alist>hash test-cont-click
  =>
HTTP/1.0 200 Document follows
Content-Type: text/html

<html>
  <head><title>Hello!</title></head>
  <body>
    <p>Chris, Good to see you!</p>
  </body>
</html>
```

As you can see the post data was sent correctly.

Chapter 15

Parsing Expression Grammars

I'm working on a new parser combinator library for Factor based on Parsing Expression Grammars and Packrat parsers. This is based on what I learnt from writing a packrat parser in Javascript.

It's progressing quite well and already fixes some problems in the existing parser combinator library I wrote. The main issue with that one is it's not tail recursive and some combinations of parsers can run out of call stack.

The new library is in the `peg` vocabulary. I haven't yet implemented the packrat side of things though so it is slow on large grammars and inputs.

I've also done a proof of concept of something I've been meaning to do for awhile. That is writing a parser for EBNF (or similar) that produces Factor code in the form of parser combinators to implement the described grammar. The code for this is in the `peg.ebnf` vocabulary. It allows you to embed an EBNF-like language and have Factor words generated for each rule:

```
<EBNF
digit  = '1' | '2' | '3' | '4' .
number = digit  digit  .
expr   = number ('+' | '-' ) number  .
EBNF>
```

This example would create three Factor words. `digit`, `number` and `expr`. These words return parsers that can be used as normal:

```
"123" number parse
"1" digit parse
"1+243+342" expr parse
```

The EBNF accepted allows for choice, zero or more repetition, optional (exactly 0 or 1), and grouping. The generated AST is pretty ugly so by default it works best as a syntax checker. You can modify the generated AST with action productions:

```
<EBNF
digit  = '1' | '2' | '3' | '4' => convert-to-digit .
number = digit  digit        => convert-to-number .
expr   = number '+' number     => convert-to-expr .
EBNF>
```

An action is a factor word after the =>. The word receives the AST produced from the rule on the stack and it can replace that with a new value that will be used in the AST. So `convert-to-expr` above might produce a tuple holding the expression values (by default, a sequence of terms in the rule are stored in a vector):

```
TUPLE: ast-expr lhs operator rhs ;
C: <ast-expr> ast-expr
: convert-to-expr ( old -- new )
  first3 <ast-expr> ;
```

The generated code is currently pretty ugly, mainly due to it being a quick proof of concept. I'll try doing a few grammars and tidy it up, changing the interface if needed, as I go along.

As an experiment I did a grammar for the PL/0 programming language. It's in `peg.pl0`. The grammar from the wikipedia article is:

```
program = block "." .

block = [ "const" ident "=" number "," ident "=" number ";"]
        [ "var" ident "," ident ";"]
         "procedure" ident ";" block ";"  statement .

statement = [ ident ":=" expression | "call" ident |
            "begin" statement ";" statement  "end" |
            "if" condition "then" statement |
            "while" condition "do" statement ].

condition = "odd" expression |
            expression ("="|"#"|"<"|"<="|">"|">=") expression .

expression = [ "+"|"-"] term  ("+"|"-") term.

term = factor ("*"|"/") factor.

factor = ident | number | "(" expression ")".
```

The Factor grammar is very similar:

```
: ident ( -- parser )
```

```
  CHAR: a CHAR: z range
  CHAR: A CHAR: Z range 2array choice repeat1
  [ >string ] action ;

: number ( -- parser )
  CHAR: 0 CHAR: 9 range repeat1 [ string>number ] action ;

<EBNF
program = block '.' .
block = [ 'const' ident '=' number { ',' ident '=' number } ';' ]
        [ 'var' ident { ',' ident } ';' ]
        { 'procedure' ident ';' [ block ';' ] } statement .
statement = [ ident ':=' expression | 'call' ident |
              'begin' statement {';' statement } 'end' |
              'if' condition 'then' statement |
              'while' condition 'do' statement ] .
condition = 'odd' expression |
            expression ('=' | '#' | '<=' | '<' | '>=' | '>') expression .
expression = ['+' | '-'] term {('+' | '-') term } .
term = factor {('*' | '/') factor } .
factor = ident | number | '(' expression ')'
EBNF>
```

This grammar as defined works and can parse PL/0 programs. I'll extend this as I improve the EBNF routines, adding actions, etc to generated a decent AST.

15.1 Parsing Arithmetic Expressions

This is it without parenthesis handling. I'll leave it as an exercise to add that. I included code to compile the ast into a factor expression. Use like:

```
"2+3*4-6/2" expr parse parse-result-ast .

"2+3*4-6/2" expr parse parse-result-ast ast-compile dup .

"2+3*4-6/2" expr parse parse-result-ast ast-compile call
```

The grammar is a direct translation of the one shown in the wikipedia peg article:

```
http://en.wikipedia.org/wiki/Parsing_expression_grammar
```

The only complication is transforming the result of using 'repeat0' into a tree by doing a fold. This is standard practice with parsing expression grammars that don't handle left recursion. When I add left recursion this won't be needed and the grammar will be simplified. See this article for details:

```
http://vpri.org/pdf/packrat_TR-2007-002.pdf

USING: sequences kernel math words math.parser namespaces peg ;
IN: scratchpad

: seq* ( quot -- parser )
 { } make seq ; inline

: choice* ( quot -- parser )
 { } make choice ; inline

TUPLE: operator lhs op rhs  ;
C: <operator> operator

: operator-fold ( lhs seq -- value )
 #! Perform a fold of a lhs, followed by a sequence of pairs being
 #! { operator rhs } in to a tree structure of the correct precedence.
 swap [ first2 <operator> ] reduce ;

: digits ( -- parser )
 CHAR: 0 CHAR: 9 range repeat1 [ string>number ] action ;

: plus ( -- parser )
 "+" token ;

: minus ( -- parser )
 "-" token ;

: multiply ( -- parser )
 "*" token ;

: divide ( -- parser )
 "/" token ;

DEFER: expr

: value ( -- parser )
 [ digits , [ expr ] delay , ] choice* ;

: product ( -- parser )
 [
   value ,
   [
     [ multiply , divide , ] choice* ,
     value ,
   ] seq* repeat0 ,
```

```
 ] seq* [ first2 operator-fold ] action ;

: sum ( -- parser )
 [
   product ,
   [
     [ plus , minus , ] choice* ,
     product ,
   ] seq* repeat0 ,
 ] seq* [ first2 operator-fold ] action ;

: expr ( -- parser )
 sum ;

GENERIC: (compile) ( ast -- )

M: number (compile) ( ast -- )
 , ;

M: operator (compile) ( ast -- )
 dup operator-lhs (compile)
 dup operator-rhs (compile)
 operator-op "math" lookup , ;

: ast-compile ( ast -- quot )
 [ (compile) ] [ ] make ;
```

Chapter 16

Factor to Javascript Compiler

Factor to Javascript compiler to be described here.

Chapter 17

Git Repository

17.1 How to publish a git repository

To set up a repository on a server you should clone the existing Factor repository using the '–bare' option:

```
git clone --bare http://www.factorcode.org/git/factor.git factor.git
```

A bare repository is one without a checked out working copy of the code. It only contains the git database. As a general rule you should never push into a repository that contains changes in the working copy. To ensure this doesn't happen, we're making the server repository a bare repository - it has no working copy.

Copy the 'factor.git' directory onto your server. I put it in '/git/factor.git'. Now if you have changes on your local machine that you want to push to your repository you can use something like:

```
git push yourname@yourserver.com:/git/factor.git
```

If you want to push changes from a specific branch in your local repository:

```
git push yourname@yourserver.com:/git/factor.git mybranch:master
```

To publish the remote repository you have two options. You can publish via the HTTP protocol, or via the git protocol. The first is slower but usable by people behind restrictive firewalls, while the second is more efficient but requires an open port. I suggest doing both.

To publish via HTTP, you must make the file 'hooks/post-update' executable:

```
chmod +x /git/factor.git/hooks/post-update
```

This gets executed whenever something is pushed to the repository. It runs a command 'git-update-server-info' which updates some files that makes the HTTP retrieval work. You should also run this once manually:

```
cd /git/factor.git
git-update-server-info
```

Now make the /git directory published via your webserver (I symbolic link to it in the server's doc-root). People can pull from the repository with:

```
git pull http://yourserver.com/git/factor.git
```

To set up the git protocol you need to run the 'git-daemon' command. You pass it a directory which is the root of your git repositories. It will make public all git repositories underneath that root that have the file 'git-daemon-export-ok' in it. So first create this file:

```
touch /git/factor.git/git-daemon-export-ok
```

Run the daemon with:

```
git-daemon --verbose /git
```

The '–verbose' will give you output showing the results of connecting to it. I run this from within a screen session. You can set it up to automatically run using whatever features your server OS has. Now people can retrieve via the git protocol:

```
git pull git://yourserver.com/git/factor.git
```

My repository is accessible from both protocols:

```
git clone http://www.double.co.nz/git/factor.git
git clone git://double.co.nz/git/factor.git
```

17.2 Binary Files

Git has a heuristic for detecting binary files. You can force other file types to be binary by adding a .gitattributes file to your repository. This file contains a list of glob patterns, followed by attributes to be applied to files matching those patterns. By adding .gitattributes to the repository all cloned repositories will pick this up as well.

For example, if you want all *.foo files to be treated as binary files you can have this line in .gitattributes:

```
*.foo -crlf -diff -merge
```

This will mean all files with a .foo extension will not have carriage return/line feed translations done, won't be diffed and merges will result in conflicts leaving the original file untouched.

Now when you pull from another repository that has changes to a .foo file you'll see something like:

```
 test.foo |  Bin 32 -> 36 bytes
```

Note that it shows it is a binary file. If you pull from another repository with changes to test.foo you'll get:

```
Auto-merged test.foo
CONFLICT (content): Merge conflict in test.foo
```

The file will be untouched and you can change it manually to be the correct version. Either by leaving it untouched, or copying a new file over it. Then you need to commit the merge conflict fix (even if you left the file untouched):

```
git commit -a -m "Fix merge conflict in test.foo"
```

17.3 Cherry Picking

The cherry picking of patches works differently to Darcs. There are a couple of ways of handling this, but I use 'git cherry-pick'. If you have a number of contributers with their own repositories that you regularly pull from you can set up remote tracking branches:

```
git remote add john http://...
git remote add mary http://...
```

Now when you want John and Mary's most recent patches you can fetch them:

```
git fetch john
git fetch mary
```

This does not make any changes to your local branches. It gets and stores their changes in a separate remote tracking branch. If you want to see what John has changed, compared to yours:

```
git log -p master..john/master
```

From there you can decide to pull in all John's commits:

```
git merge john/master
```

If you want one commit, but not its dependencies then this is where 'cherry-pick' is used.

Given a commit id, 'cherry-pick' will take the patch for that commit and apply it to your current branch. It's used like:

```
git cherry-pick abcdefgh
```

This creates a commit with a different commit id than the original, but with the same contents. It needs to be a different id as it doesn't have the same dependencies as the original.

If you decide later you want all John's commits and do a merge which includes the commit that you cherry picked from you might expect conflicts. Git handles this case fine and does an automatic merge, noticing the patches are the same. So it effectively gives you the same functionality as Darcs selective patch pulling, but not as nice a user interface.

Chapter 18

Ogg Vorbis and Theora

Describe Ogg library usage.

Chapter 19

Serialization

19.1 Serializing Objects

Serialisation supports most Factor types and is extensible by adding methods to generic functions. You can even serialize quotations containing words, and so long as those words exist in the target system, the deserialization will link them correctly and the quotation is callable. Actual code doesn't serialize at this stage though.

An example of usage:

```
"serialize" require
USE: serialize

[
  [ "Hello World!" serialize ] with-serialized
] string-out
  => ...serialized format as a string...

[
  [ deserialize ] with-serialized
] string-in
  => "Hello World!"
```

The 'string-out' and 'string-in' shown above are standard words to direct all output to go to and from strings respectively. You can also serialize to files and deserialize them on any other Factor system.

19.2 Serializing Gadgets

I've managed to get the serialization code to the point where it serializes user interface gadgets. To demonstrate it I serialized an in-progress game of Space Invaders, uploaded the serialized file to my web server and someone else on IRC downloaded it, deserialized it, and continued where the game left off.

Given the gadget on the stack, serialization to a file was as easy as:

```
[
  "filename.ser" <file-writer> [
    unparent serialize
  ] with-stream
] with-serialized
```

To get the instance running again:

```
[
  "filename.ser" <file-writer> [
    deserialize
  ] with-stream
] with-serialized "Space Invaders" open-titled-window
```

Index